Pipelines

Tyler Bletsch
Duke University

Slides are derived from work by Andrew Hilton (Duke) and Amir Roth (Penn)
Clock Period and CPI

• Single-cycle datapath
  • Low CPI: 1
  • Long clock period: to accommodate slowest insn

• Multi-cycle datapath
  • Short clock period
  • High CPI

• Can we have both low CPI and short clock period?
  • No good way to make a single insn go faster
  • Insn latency doesn’t matter anyway ... insn throughput matters
  • Key: exploit inter-insn parallelism
Remember The von Neumann Model?

- **Instruction Fetch**: Read instruction bits from memory
- **Decode**: Figure out what those bits mean
- **Operand Fetch**: Read registers (+ mem to get sources)
- **Execute**: Do the actual operation (e.g., add the #s)
- **Result Store**: Write result to register or memory
- **Next Instruction**: Figure out mem addr of next insn, repeat

We’ll call this the “VN loop”
Pipelining

- **Pipelining**: important performance technique
  - Improves insn throughput rather than insn latency
  - Exploits parallelism at insn-stage level to do so
  - Begin with multi-cycle design

  ![Pipeline Diagram]

  - When insn advances from stage 1 to 2, next insn enters stage 1

  ![Pipeline Diagram]

  - Individual insns take same number of stages
  - **But insns enter and leave at a much faster rate**
  - Physically breaks “atomic” VN loop ... but must maintain illusion

- Automotive assembly line analogy
5 Stage Pipelined Datapath

- Temporary values (PC, IR, A, B, O, D) re-latched every stage
  - Why? 5 insns may be in pipeline at once, they share a single PC?
  - Notice, PC not latched after ALU stage (why not?)
Pipeline Terminology

- Stages: **Fetch**, **Decode**, **eXecute**, **Memory**, **Writeback**
- Latches (pipeline registers): **PC**, **F/D**, **D/X**, **X/M**, **M/W**
Some More Terminology

- **Scalar pipeline**: one insn per stage per cycle
  - Alternative: “superscalar” (take 552)

- **In-order pipeline**: insns enter execute stage in VN order
  - Alternative: “out-of-order” (take 552)

- **Pipeline depth**: number of pipeline stages
  - Nothing magical about five
  - Trend has been to deeper pipelines
Pipeline Example: Cycle 1

- 3 instructions

add $3,$2,$1
Pipeline Example: Cycle 2

lw $4,0($5)  add $3,$2,$1

- 3 instructions
**Pipeline Example: Cycle 3**

- 3 instructions

```plaintext
sw $6,4($7)  lw $4,0($5)  add $3,$2,$1
```
Pipeline Example: Cycle 4

- 3 instructions
Pipeline Example: Cycle 5

- 3 instructions

sw $6,4($7)  lw $4,0($5)  add
Pipeline Example: Cycle 6

- 3 instructions
• 3 instructions
**Pipeline Diagram**

- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: insns
  - Convention: X means `lw $4,0($5)` finishes execute stage and writes into X/M latch at end of cycle 4

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add $3,$2,$1</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw $4,0($5)</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sw $6,4($7)</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What About Pipelined Control?

• Should it be like single-cycle control?
  • But individual insn signals must be staged

• How many different control units do we need?
  • One for each insn in pipeline?

• Solution: use simple single-cycle control, but pipeline it
  • Single controller
    • Key idea: pass control signals with instruction through pipeline
Pipelined Control

Diagram showing the pipeline stages of an instruction execution process, including PC, Register File, Data Memory, and CTRL.
Pipeline Performance Calculation

• Single-cycle
  • Clock period = 50ns, CPI = 1
  • Performance = 50ns/insn

• Multi-cycle
  • Branch: 20% (3 cycles), load: 20% (5 cycles), other: 60% (4 cycles)
  • Clock period = 12ns, CPI = (0.2*3+0.2*5+0.6*4) = 4
    • Remember: latching overhead makes it 12, not 10
  • Performance = 48ns/insn

• Pipelined
  • Clock period = 12ns
  • CPI = 1.5 (on average insn completes every 1.5 cycles)
  • Performance = 18ns/insn
Some questions (1)

- **Why Is Pipeline Clock Period > delay thru datapath / number of pipeline stages?**
  - Latches (FFs) add delay
  - Pipeline stages have different delays, clock period is max delay
  - Both factors have implications for ideal number pipeline stages
Some questions (2)

• **Why Is Pipeline CPI > 1?**
  • CPI for scalar in-order pipeline is **1 + stall penalties**
  • Stalls used to resolve hazards
    • **Hazard**: condition that jeopardizes VN illusion
    • **Stall**: artificial pipeline delay introduced to restore VN illusion

• **Calculating pipeline CPI**
  • **Frequency of stall * stall cycles**
  • Penalties add (stalls generally don’t overlap in in-order pipelines)
  • $1 + \text{stall-freq}_1 \times \text{stall-cyc}_1 + \text{stall-freq}_2 \times \text{stall-cyc}_2 + ...$

• **Correctness/performance/MCCF**
  • Long penalties OK if they happen rarely, e.g., $1 + 0.01 \times 10 = 1.1$
  • Stalls also have implications for ideal number of pipeline stages
Dependences and Hazards

• **Dependence**: relationship between two insns
  - **Data**: two insns use same storage location
  - **Control**: one insn affects whether another executes at all
  - Not a bad thing, programs would be boring without them
  - Enforced by making older insn go before younger one
    - Happens naturally in single-/multi-cycle designs
    - But not in a pipeline

• **Hazard**: dependence & possibility of wrong insn order
  - Effects of wrong insn order cannot be externally visible
    - **Stall**: for order by keeping younger insn in same stage
  - Hazards are a bad thing: stalls reduce performance
Why Does EveryInsn Take 5 Cycles?

- Could /should we allow `add` to skip M and go to W? No
  - It wouldn’t help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: imagine `add` follows `lw`

![Diagram of the processor's pipeline stages with PC,Insn Mem, Register File, Data Mem, and execution stages labeled as PC, IR, O, D, X, M/W, F/D, A, B, S, X. The diagram illustrates the flow of instructions and data through the pipeline, with operations such as `add $3,$2,$1` and `lw $4,0($5)` highlighted.](image-url)
Structural Hazards

• **Structural hazards**
  - Two insns trying to use same circuit at same time
    • E.g., structural hazard on regfile write port

• **To fix structural hazards**: proper ISA/pipeline design
  - Each insn uses every structure exactly once
  - For at most one cycle
  - Always at same stage relative to F
Let’s forget about branches and the control for a while

The three insn sequence we saw earlier executed fine...
  • But it wasn’t a real program
  • Real programs have data dependences
    • They pass values via registers and memory
Data Hazards

- Would this “program” execute correctly on this pipeline?
  - Which insns would execute with correct inputs?
  - `add` is writing its result into `$3` in current cycle
    - `lw` read `$3` 2 cycles ago → got wrong value
    - `addi` read `$3` 1 cycle ago → got wrong value
  - `sw` is reading `$3` this cycle → OK (regfile timing: write first half)
Memory Data Hazards

- What about data hazards through memory? No
  - \texttt{lw} following \texttt{sw} to same address in next cycle, gets right value
  - Why? DMem read/write take place in same stage

- Data hazards through registers? Yes (previous slide)
  - Occur because register write is 3 stages after register read
  - Can only read a register value 3 cycles after writing it
Fixing Register Data Hazards

• Can only read register value 3 cycles after writing it

• One way to enforce this: make sure programs don’t do it
  • Compiler puts two independent insns between write/read insn pair
    • If they aren’t there already
  • Independent means: “do not interfere with register in question”
    • Do not write it: otherwise meaning of program changes
    • Do not read it: otherwise create new data hazard
  • Code scheduling: compiler moves around existing insns to do this
  • If none can be found, must use nops

• This is called software interlocks
  • MIPS: Microprocessor w/out Interlocking Pipeline Stages
Software Interlock Example

```
sub $3,$2,$1
lw $4,0($3)
sw $7,0($3)
add $6,$2,$8
addi $3,$5,4
```

- Can any of last 3 insns be scheduled between first two?
  - `sw $7,0($3)`? No, creates hazard with `sub $3,$2,$1`
  - `add $6,$2,$8`? OK
  - `addi $3,$5,4`? YES...-ish. Technically. (but it hurts to think about)
    - Would work, since `lw` wouldn’t get its $3 from it due to delay
    - Makes code REALLY hard to follow – each instruction’s effects “happen” at different delays (memory writes “immediate”, register writes delayed, etc.)
    - Let’s not do this, and just add a `nop` where needed
  - Still need one more insn, use `nop`
    - `add $3,$2,$1`
    - `add $6,$2,$8`
    - `nop`
    - `lw $4,0($3)`
    - `sw $7,0($3)`
    - `addi $3,$5,4`
Software Interlock Performance

• Same deal
  • Branch: 20%, load: 20%, store: 10%, other: 50%

• Software interlocks
  • 20% of insns require insertion of 1 \texttt{nop}
  • 5% of insns require insertion of 2 \texttt{nops}

• CPI is still 1 technically
• But now there are more insns
• \#insns = 1 + 0.20*1 + 0.05*2 = 1.3
  - 30% more insns (30% slowdown) due to data hazards
Hardware Interlocks

• Problem with software interlocks? Not compatible
  • Where does 3 in “read register 3 cycles after writing” come from?
    • From structure (depth) of pipeline
  • What if next MIPS version uses a 7 stage pipeline?
    • Programs compiled assuming 5 stage pipeline will break

• A better (more compatible) way: **hardware interlocks**
  • Processor detects data hazards and fixes them
  • Two aspects to this
    • Detecting hazards
    • Fixing hazards
Detecting Data Hazards

- Compare F/D insn input register names with output register names of older insns in pipeline
  - Hazard =
    - (F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
    - (F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)
Fixing Data Hazards

- Prevent F/D insn from reading (advancing) this cycle
  - Write `nop` into D/X.IR (effectively, insert `nop` in hardware)
  - Also reset (clear) the datapath control signals
  - Disable F/D latch and PC write enables (why?)
- Re-evaluate situation next cycle
Hardware Interlock Example: cycle 1

(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

= 1
Hardware Interlock Example: cycle 2

(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

= 1
Hardware Interlock Example: cycle 3

\[(F/D.IR.RS1 == D/X.IR.RD) \lor (F/D.IR.RS2 == D/X.IR.RD) \lor \\
(F/D.IR.RS1 == X/M.IR.RD) \lor (F/D.IR.RS2 == X/M.IR.RD)\] = 0

\[1w \ $4,0(\$3) \]

\[\text{add } \$3,\$2,\$1\]
Pipeline Control Terminology

- Hardware interlock maneuver is called **stall** or **bubble**

- Mechanism is called **stall logic**

- Part of more general **pipeline control** mechanism
  - Controls advancement of insns through pipeline

- Distinguished from **pipelined datapath control**
  - Controls datapath at each stage
  - Pipeline control controls advancement of datapath control
Pipeline Diagram with Data Hazards

- Data hazard stall indicated with d*
  - Stall propagates to younger insns

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $3,$2,$1</td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $4,0($3)</td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $6,4($7)</td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- This is not OK (why?)
Hardware Interlock Performance

- Hardware interlocks: same as software interlocks
  - 20% of insns require 1 cycle stall (i.e., insertion of 1 \texttt{nop})
  - 5% of insns require 2 cycle stall (i.e., insertion of 2 \texttt{nops})

- CPI = 1 + 0.20*1 + 0.05*2 = 1.3
- So, either CPI stays at 1 and \#insns increases 30% (software)
- Or, \#insns stays at 1 (relative) and CPI increases 30% (hardware)
- Same difference

- Anyway, we can do better
Observe

- This situation seems broken
  - \texttt{lw \ $4,0(\$3)} has already read $3$ from regfile
  - \texttt{add \ $3,\$2,\$1} hasn’t yet written $3$ to regfile
- But fundamentally, everything is still OK
  - \texttt{lw \ $4,0(\$3)} hasn’t actually \textcolor{red}{used} $3$ yet
  - \texttt{add \ $3,\$2,\$1} has already computed $3$
• **Bypassing**
  - Reading a value from an intermediate (μarchitectural) source
  - Not waiting until it is available from primary source (RegFile)
  - Here, we are bypassing the register file
  - Also called **forwarding**
• What about this combination?
  • Add another bypass path and MUX input
  • First one was an **MX** bypass
  • This one is a **WX** bypass

```assembly
lw $4,0($3)  add $3,$2,$1
```

**WX Bypassing**
ALUinB Bypassing

• Can also bypass to ALU input B
Does WM bypassing make sense?

- Not to the address input (why not?)
  - Address input requires the ALU to compute; value is not ready *anywhere* in the CPU
- But to the store data input, yes
• Each MUX has its own, here it is for MUX ALUinA
  
  \[(D/X.IR.RS1 == X/M.IR.RD) \rightarrow \text{mux select} = 0\]
  
  \[(D/X.IR.RS1 == M/W.IR.RD) \rightarrow \text{mux select} = 1\]
  
  Else \(\rightarrow\) mux select = 2
Bypass and Stall Logic

- Two separate things
  - Stall logic controls pipeline registers
  - Bypass logic controls muxes

- But complementary
  - For a given data hazard: if can’t bypass, must stall

- Slide #43 shows **full bypassing**: all bypasses possible
  - Is stall logic still necessary? Yes
Stall = (D/X.IR.OP==LOAD) && (F/D.IR.RS1==D/X.IR.RD) || ((F/D.IR.RS2==D/X.IR.RD) && (F/D.IR.OP!=STORE))

Intuition: “Stall if it’s a load where rs1 is a data hazard for the next instruction, or where rs2 is a data hazard in a non-store next instruction”. This is because rs2 is safe in a store instruction, because it doesn’t use the X stage, and can be M/W bypassed.
Pipeline Diagram With Bypassing

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $3,$2,$1</code></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>lw $4,0($3)</code></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addi $6,$4,1</code></td>
<td>F</td>
<td>D</td>
<td>d*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sometimes you will see it like this
  - Denotes that stall logic implemented at X stage, rather than D
  - Equivalent, doesn’t matter when you stall as long as you do
- **Control hazards**
  - Must fetch post branch insns before branch outcome is known
  - Default: assume “not-taken” (at fetch, can’t tell it’s a branch)
**Branch recovery**: what to do when branch is taken

- **Flush** insns currently in F/D and D/X (they’re wrong)
  - Replace with **NOPs**
  - Haven’t yet written to permanent state (RegFile, DMem)
### Branch Recovery Pipeline Diagram

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $3,$0,1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnez $3,$targ</td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $6,4($7)</td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $8,$7,1</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>targ: sw $6,4($7)</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Control hazards indicated with $c^*$
- Penalty for taken branch is 2 cycles
Branch Performance

• Again, measure effect on CPI (clock period is fixed)

• Back of the envelope calculation
  • **Branch: 20%**, load: 20%, store: 10%, other: 50%
  • 75% of branches are taken (why so many taken?)

• CPI if no branches = 1
• CPI with branches = 1 + 0.20*0.75*2 = 1.3
  – **Branches cause 30% slowdown**
• How do we reduce this penalty?
Fast Branch

- **Fast branch**: can decide at D, not X
  - Test must be comparison to zero or equality, no time for ALU
    - New taken branch penalty is 1
    - Additional insns (\texttt{slt}) for more complex tests, must bypass to D too
  - 25% of branches have complex tests that require extra insn

\[
\text{CPI} = 1 + 0.20 \times 0.75 \times 1(\text{branch}) + 0.20 \times 0.25 \times 1(\text{extra insn}) = 1.2
\]
Speculative Execution

- Speculation: “risky transactions on chance of profit”

- **Speculative execution**
  - Execute before all parameters known with certainty
  - **Correct speculation**
    - Avoid stall, improve performance
  - **Incorrect speculation (mis-speculation)**
    - Must abort/flush/squash incorrect insns
    - Must undo incorrect changes (recover pre-speculation state)
  - The “game”: \[ \%_{\text{correct}} \times \text{gain} \] – \[ (1-\%_{\text{correct}}) \times \text{penalty} \]

- **Control speculation**: speculation aimed at control hazards
  - Unknown parameter: are these the correct insns to execute next?
Control Speculation Mechanics

• Guess branch target, start fetching at guessed position
  • Doing nothing is implicitly guessing target is PC+4
  • Can actively guess other targets: dynamic branch prediction

• Execute branch to verify (check) guess
  • Correct speculation? keep going
  • Mis-speculation? Flush mis-speculated insns
    • Hopefully haven’t modified permanent state (Regfile, DMem)
      + Happens naturally in in-order 5-stage pipeline

• “Game” for in-order 5 stage pipeline
  • $\%_{\text{correct}} = ?$
  • Gain = 2 cycles
    + Penalty = 0 cycles $\rightarrow$ mis-speculation no worse than stalling
Dynamic Branch Prediction

- **Dynamic branch prediction**: guess outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction** (notice new recovery circuit)
Branch Prediction: Short Summary

- **Key principle of micro-architecture:**
  - Programs do the same thing over and over (why?)
  - Exploit for performance:
    - Learn what a program did before
    - Guess that it will do the same thing again

- **Inside a branch predictor: the short version**
  - Use some of the PC bits as an **index** to a separate RAM
  - This RAM contains (a) branch destination and (b) whether we predict the branch will be taken
  - RAM is updated with results of past executions of branches
  - Algorithm for predictions can be simple (“assume it’s same as last time”), or get quite fancy
Branch Prediction Performance

- Same parameters
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken

- Dynamic branch prediction
  - Assume branches predicted with 75% accuracy (so 25% are penalized)
  - CPI = 1 + 0.20*0.25*2 = 1.1

- Branch (esp. direction) prediction was a hot research topic
  - Accuracies now 90-95%
Pipelining And Exceptions

- Remember exceptions?
  - Pipelining makes them nasty

- 5 instructions in pipeline at once

- Exception happens, how do you know which instruction caused it?
  - Exceptions propagate along pipeline in latches
  - Two exceptions happen, how do you know which one to take first?
  - One belonging to oldest insn
  - When handling exception, have to flush younger insns
    - Piggy-back on branch mis-prediction machinery to do this

- Just FYI – we’ll solve this problem in ECE 552 (CS 550)
Pipeline Depth

- No magic about 5 stages, trend had been to deeper pipelines
  - 486: 5 stages (50+ gate delays / clock)
  - Pentium: 7 stages
  - Pentium II/III: 12 stages
  - Pentium 4: 22 stages (~10 gate delays / clock) “super-pipelining”
  - Core1/2: 14 stages

- Increasing pipeline depth
  + Increases clock frequency (reduces period)
  − But decreases IPC (increases CPI)
  - Branch mis-prediction penalty becomes longer
  - Non-bypassed data hazard stalls become longer
  - At some point, CPI losses offset clock gains, question is when?
    - 1GHz Pentium 4 was slower than 800 MHz PentiumIII
  - What was the point? People by frequency, not frequency * IPC
Real pipelines…

• Real pipelines fancier than what we have seen
  • Superscalar: multiple instructions in a stage at once
  • Out-of-order: re-order instructions to reduce stalls
  • SMT: execute multiple threads at once on processor
    • Side by side, sharing pipeline resources
  • Multi-core: multiple pipelines on chip
    • Cache coherence: No stale data